Minimizing Program Startup Time by I/O Activity Mining

JinKyu Jung
jkjung@camars.kaist.ac.kr
Computer Science Div.,
EECS Dept., KAIST, KOREA

Minkoo Seo
mkseo@bulsai.kaist.ac.kr
Computer Science Div.,
EECS Dept., KAIST, KOREA

Abstract

Disk prefetching has been widely accepted as a way of minimizing process blocking time due to I/Os. However, most of current techniques highly depend on manual labors or source program analyses, thereby making prefetching less attractive and less applicable. Hence, in this paper, we propose disk prefetching based on rule based mining approach. In this scheme, we assume that I/Os happen during program startups are easily predictable because there are lots of configuration files read by programs in the initial loading stage. Therefore, we made such predictable I/Os to be analyzed and automatically prefetched by the kernel. To evaluate the performance of the proposed method, we conducted extensive experiments using a synthesized application and server programs like Apache and Tomcat. According to the experimental results, it was shown that the proposed approach reduced program startup time up to 70% with synthesized application, and up to 7% with real application.

1 Introduction

Disk I/O prefetching techniques are widely investigated as a way of minimizing process blocking time [3, 4, 5, 7]. The rationale behind prefetching is as follows: If disk blocks could be loaded in advance, it will save time needed for disk I/O that will happen in the future. Such a idea was widely accepted, and made system call like *readahead* installed by default.

However, current operating systems do not have any way of automatically applying readahead system calls. To alleviate the problem, exploiting user supplied disclose [6], compiler based automatic prefetch code insertion [4], mining based algorithm [5] were proposed. However, each of these approaches has serious drawbacks. Firstly, user supplied disclose that describes what will be read in the future lay too much burden on programmers. Secondly, compiler based algorithms need heavy modification of compilers and the source code itself which is not always provided to the software users. Thirdly, previous mining based algorithms employ CPU intensive complex mining algorithms that might not be applicable always. Therefore, we propose a disk prefetching technique which applies simple data mining techniques for automatically applying readahead system calls when a program starts.

In the proposed scheme, it is assumed that the almost same configuration files are loaded whenever program starts. According to our preliminary experiments it was shown that the assumption holds. For example, one of the famous text editor, *VI*, reads /etc/vimrc, /usr/share/vim/colors/*, /usr/share/vim/syntax/*, etc whoever started the program. Hence, proposed scheme works as follows. TxMonitor monitors all I/O activities and delivers that information to IOMiner. When CPU is idle, the miner generates I/O rules and pass them to the DiskPrefetcher. Later, whenever a program starts, DiskPrefetcher does readahead system calls automatically on behalf of the program.

The remainder of the paper is organized as follows. Chapter 2 introduces related works and highlights the differences between past works and ours. Chapter 3 depicts overall system architecture. Chapter 4, 5, and 6 explains TxMonitor, IOMiner, and DiskPrefetcher, respectively. Chapter 7 presents experimental results, and chapter 8 concludes this paper.

2 Related Work

Kotz tried to detecting more complex pattern within in a file than the sequential ones [3]. In the paper, some patterns for local prefetching and global prefetching were designed. According to predefined patterns, files were prefetched. Though the algorithm was shown to be beneficial in most of time, the patterns proposed were heuristic at best and lack theoretical foundation.

Mowry et al. analyzed matrix operations and inserted prefetch operations automatically by compilers [4]. However, their algorithm need source code of a program. In addition, it is not always possible to to modify compilers commercially used.

Patterson et al. [7] devised a formula which can be used to compare pros and cons of prefetching and caching. Based on the formula, the decision whether to prefetch or to cache was made. To determine what to prefetch, their algorithm heavily depends on user supplying disclose [6] which describes what will be read in the future.

These days, traditional prefetching algorithms are being applied to network I/O. Nanopoulous et al. presented web prefetching algorithm based on Markov predictor [5]. In their approach, a server which implemented prediction algorithm is assumed. And that server was responsible for piggybacking prefetching rules. The rules generated was based on the probabilistic correlation of document accesses. To the contrary, in this paper, we only analyze disk I/O activities that follows a program execution, thereby making expensive probabilistic model like Markov predictor unnecessary.

3 System Architecture

Our system comprises three components: TxMonitor, IOMiner, and DiskPrefetcher. Figure 1 shows the overall architecture.

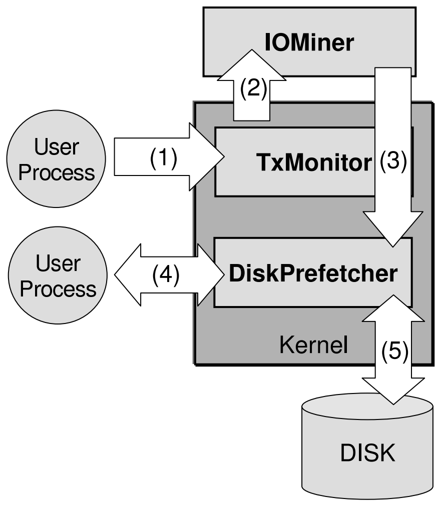

Figure 1: System architecture and the processing flow of proposed system

The system operates as follows. (1) When I/O transaction monitoring is enabled and a program starts, I/O activities of that program are recorded by TxMonitor. (2) The activities are passed to IOMiner when the CPU is idle. (3) IOMiner analyze the I/O activities and finds initial I/O burst that loads all configuration files. (4) Later, when the same program starts, (5) DiskPrefetcher does all readahead required for speeding up that program.

4 TxMonitor

TxMonitor gathers all I/O transactions are explained, and that information has to be in the form that is well suited for identifying initial IO burst.

For that purpose, TxMonitor records all program startups and I/O read operations. On one hand, when a program starts, just the name of the program is recorded. On the other hand, when a program perform read operation, it is needed to records (1) the name of program that performed read, (2) name of data file that was read, (3) starting position in the data file, (4) length of data read, (5) time when the read operation was done.

For example, table 1 shows some parts of I/Os that was happened after VI started.[1]

Table 1: Example IO activities of VI recorded by TxMonitor.

start/stop:1 /bin/vi
start:128 length:826 vi /usr/.../xterm
start:954 length:1318 vi /usr/.../xterm
start:2272 length:10 vi /usr/.../xterm
start:0 length:4096 vi /etc/vimrc
start:0 length:4096 vi /usr/.../evening.vim
start:2381 length:4096 vi /usr/.../evening.vim
start:2381 length:4096 vi /usr/.../evening.vim
start:0 length:4096 vi /usr/.../syntax.vim
start:0 length:4096 vi /usr/.../synload.vim
start:0 length:4096 vi /usr/.../evening.vim
start:0 length:4096 vi /usr/.../syncolor.vim
start:4093 length:4096 vi /usr/.../syncolor.vim
start:0 length:4096 vi /usr/.../syncolor.vim
start:4093 length:4096 vi /usr/.../syncolor.vim
start:2381 length:4096 vi /usr/.../evening.vim
start:1534 length:4096 vi /usr/.../synload.vim
start:0 length:4096 vi /usr/.../filetype.vim
...

Gathering program starts and I/O operations adds very little overhead according to our preliminary experiments. However, we added a system call for turning on and off TxMonitor to entirely avoid unnecessary overheads.

5 IOMiner

IOMiner analyzes I/O activities recorded by TxMonitor, and produce rules that answers the question of *what is the initial I/O activities that are done when the program ABC starts?* This kind of mining is called association rule mining [1]. However, traditional association rule mining is not directly applicable to the above problem, because they did try to answer the problem of *what will happen when X, Y, and Z happened already?* In detail, previous works tried to find one activity that follows several activities, whereas we're trying to find several activities that follows one activity. Therefore, we developed a new mining algorithm.

In our scheme, a sliding window of fixed length is used. The window is placed where a program start is recorded. Then, all I/O activities within the window is gathered.

In detail, let $X(S)_n$ denote the startup of program X which started at time n. Also, let $X(\alpha)_n$ denote a read operation that read data α by program X at time n. For example, a operation sequence $A(S)_0$, $A(\alpha)_1$, $B(S)_2$, $A(\beta)_3$, $C(S)_3$, $B(\gamma)_4$ means that 'Program A started at time 0. Then, A read α at time 1. After that, program B started at 2. A read β at 3. Then, program C started at 3. Finally, program B read γ at time 4.' Note that program start of C and B's first read operation recorded as if they happened simultaneously. This is possible because we record each event in secs.

Given a operation sequence, a sliding window of fixed length is placed on every program startups. Then, all I/O activities within the window is considered to be the *candidate set* of initial I/O activity. If the length of sliding window is 3, for example, then we get $\{A(\alpha)_1, A(\beta)_3\}$ as the initial I/O burst of A. In case of program B, we get $\{B(\gamma)_4\}$.

The candidate set generated from the windowing is not guaranteed to be the same always. This is due to the files that is specific to user accounts. For example, VI reads .vimrc resides in the user home directory, and it resolves into different file depending on the user account who executed VI. To grasp common configuration files, while removing user dependent configurations, we apply the idea of *confidence* [2].

Let X_n (where $n = 1, 2, \ldots, N$) denote a candidate set of initial I/O activity of program X. Note that there might be several candidate sets because we get a candidate set whenever X starts and TxMonitor is turned on. The confidence value of an element in the set X_n, say $A(\alpha)$, is computed as follows:

$$confidence(A(\alpha)) = \frac{\#\, of\ X_n\ that\ contains\ A(\alpha)}{N}$$

After computing confidence value of each element in the candidate set, we remove an element if its confidence value does not exceeds the predefined threshold. A rule that passed confidence test is turned into the following *ruleset*: $\{X(S) \Rightarrow X(\alpha), X(\beta), X(\gamma)\}^+$. Finally, for the sake of performance, rules in the set are sorted according to program name, X, and passed to DiskPrefetcher using a system call.

6 DiskPrefetcher

Given a rule set $\{X(S) \Rightarrow X(\alpha), X(\beta), X(\gamma)\}^+$ passed by IOMiner, DiskPrefetcher readahead data file α, β, and γ whenever program X starts.

In detail, when a program X starts, a system call, execve is called. Then, the execve is catched, and DiskPrefetcher looks up ruleset. For speeding up rule lookup, a binary search is performed. When the rule is found, each data file in the rule is opened, and prefetched into the buffer cache.

7 Performance Evaluation

7.1 Experimental Environment

We implemented proposed prefetching technique in Linux Kernel 2.6.11. TxMonitor and DiskPrefetcher were implemented in the kernel, and IOMiner was programmed as a daemon runs in user level. All programs run on Intel Pentium 630 CPU(3.0GHz), 180Gbyte SATA2 7200RPM hard disk, and 1Gbyte main memory.

In the first three experiments, we compared performance of proposed scheme with varying CPU time, number of data files, and data file size, using synthesized application. Here, CPU time represents the computation time between successive reads in the program. The last experiment run the four widely used applications, VI, Apache, Mysql, and Tomcat, with and without prefetching.

7.2 CPU Time

In this section, performance enhancement with different CPU time was measured. The number of data files were fiexed as 20, and the sizze of each file was also fixed as 10KB. Synthesized user program was programmed to read data file using 4KB buffers. Figure 2 shows the results. X axis in the figure represents the number of random() function calls that has been done to emulate some computations between reads.

According to the results, prefetching enhanced program startup time by fixed amount. However, the difference was the same whether the CPU time is long or short. This is due to the fact read operations do not block

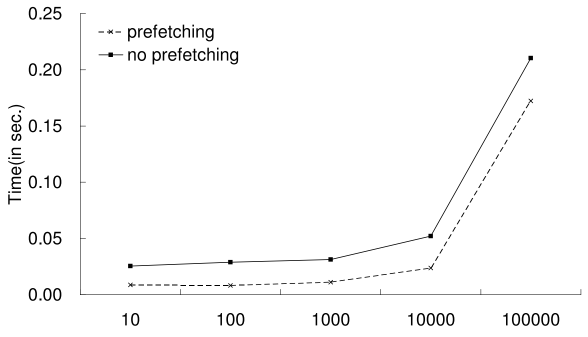

Figure 2: Performance comparison with various CPU time

program for a long time, because it does not take long time to read 4KB data.

7.3 Number of Data Files

In this section, we changed the number of data files and measured program startup time with and without prefetching. The size of each data file was fixed as 10KB. The size of file was determined as such, because data files of applications like VI, Apache, and Tomcat were about 10KB.

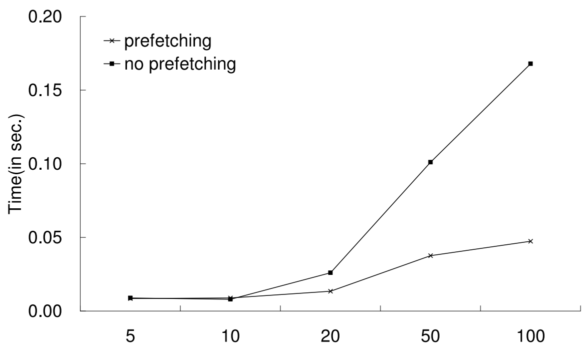

Figure 3: Performance comparison with various number of data files

According to the experimental result shown in the figure 3, prefetching improved program performance as the number of files increases. In detail, when there are 100 data files, 70% of startup time was reduced. These improvements were made because the possibility of process blocks increased as the number of files increase.

7.4 Data File Size

In this experiment, data files of 500Bytes, 1KB, 10KB, 20KB, 50KB, and 100KB were used. The number of data file was fixed as 20. Figure 4 depicts the result.

As shown in the figure 4, performance was improved more as the size of data file increases. In non prefetching version, $data\ file\ size$ / $buffer\ size$ read operations are needed. To the contrary, when prefetching is applied, only $number\ of\ data\ files$ read operations are performed. Hence, the number of random I/O decreases a lot, enhancing overall performance.

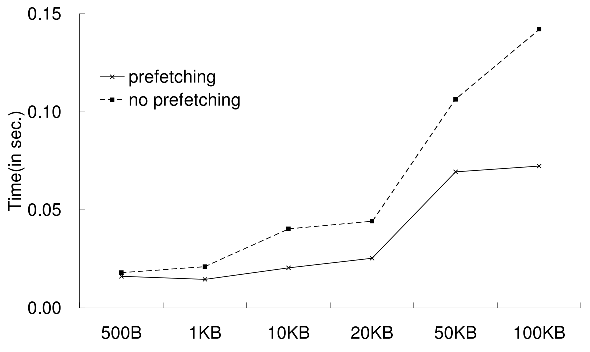

Figure 4: Performance comparison with various data file size

7.5 Real Applications

The last experiment assessed the performance enhancement using four widely used applications: VI, Apache, Mysql, and Tomcat. Figure 5 shows the performance enhancement wit prefetching. In the figure, more darker bars, above which program names are printed, represents the program startup time without prefetching, while more lighter colors represents the startup time with prefetching.

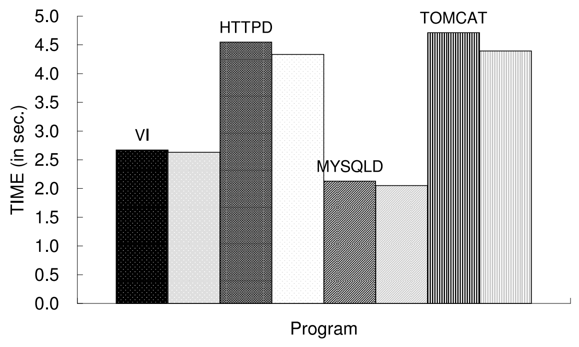

Figure 5: Performance comparison with real applications

As shown in the figure, prefetching decreased the program startup time. In case of Tomcat, for example, decreased 7% of startup time. Specifically, it is worth noting that the size of data file and the number of data files played an important role in performance enhancement.

8 Conclusion

In this paper, we proposed an I/O activity mining based program startup time minimizing scheme. In the proposed scheme, TxMonitor gathered process start and I/O read operation activities. This information was passed to IOMiner which generated IO rules. When a program starts, DiskPrefetcher prefetched data files according to the rules. According to the experimental results, program startup time decreased up to 7%.

In the future, it is needed to differentiate disk reads that blocks a process and that does not block the process. By doing this, it will be possible to generate more intelligent rules. In addition, it also might be possible to consider the number of page references that were loaded by prefetcher, thereby enhancing cache replacement strategy.

References

[1] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules”, *In Proc. of VLDB Conf.*, 1994.

[2] J. Hipp, U. Guntzer, and G. Nakhaeizadeh, “Algorithms for Association Rule Mining - A General Survey and Comparison”, *SIGKDD*, 2000.

[3] D. Kotz, C. S. Ellis, “Practical Prefetching Techniques for Parallel File Systems”, *In Proc. First International Conf. on Parallel and Distributed Information Systems*, 1991.

[4] T. C. Mowry, M. S. Lam, and A. Gupta, “Design and Evaluation of a Complier Algorithm for Prefetching”, *ASPLOS*, 1992.

[5] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos, “A Data Mining Algorithm for Generalized Web Prefetching”, *IEEE Trans. on Knowledge and DAta Engineering*, 2003.

[6] R. H. Patterson, G. Gibson, and M. Satyanarayanan, “A Status Report on Research in Transparent Informed Prefetching”, *ACM Operating Systems Review*, 1993.

[7] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, “Informed Prefetching and Caching”, *In Proc. of ACM Symp. on Operating System Principles*, 1995.

Notes

[1]Time column was omitted due to the lack of margin in the paper.